

Computer Generation of Random Variables Based on Transformation Method

Zexi Huang

Yingcai Honors College

University of Electronic Science and Technology of China

I. INTRODUCTION

Generating any random variables given their probability density function (PDF) or probability mass function (PMF) is essential in computer-based signal analysis and processing. Here, we use the transformation method to generate the required random variables and verify their relative properties. The remainder of this report is organized as follows: In the following section, we briefly introduce the method of uniformly distributed pseudo-random numbers. [Section III](#) presents method for generating binomial random variables using transformation method, verify their validity and compare the efficiency between several modified versions of algorithm. In [Section IV](#), we use a special form of transformation method to generate two independent normal random variables and test their normality and independence. Finally, we give a short discussion and conclude in [Section V](#).

Prior to detailed introduction, it is worth mentioning that the working environment is *MATLAB 9.0.0.341360 (R2016a)*, and all source codes (except for *MT19937*, which is coded in C++) included in the appendix are of that syntax.

II. UNIFORM RANDOM VARIABLES

The implementation of transformation method is based on uniform random variables between $[0, 1]$ and the given cumulated distribution function (CDF). Thus the first step of generating any type of random variables is to find a uniform random variable generator.

A. Classic Generators

The most classic type of pseudo-random number generator is the linear congruential generator (LCG), which is defined by the recurrent relation

$$X_{n+1} = (aX_n + c) \mod m \quad (1)$$

where X is the sequence of pseudorandom values, and $m > 0$, the modulus, $0 < a < m$, the multiplier, $0 \leq c < m$, the increment, $0 \leq X_0 < m$, the seed, are integer constants that specify the generator. If $c = 0$, the generator is often called a multiplicative congruential generator (MCG), or Lehmer RNG. If $c \neq 0$, the method is called a mixed congruential generator.

The LCG is very easy to implement and require minimal memory (typically 32 or 64 bits) to retain state. However, its period length is very limited (at best m , which is often $m = 2^{32}$ and $m = 2^{64}$), and it is predictable from a subsequence because of the serial correlation and thus not suitable for Monte Carlo simulation and cryptography applications.

B. Mersenne Twister Generator

The Mersenne Twister (MT) algorithm is based on a matrix linear recurrence over a finite binary field F_2 . The algorithm is a twisted generalised feedback shift register (twisted GFSR, or TGFSR) of rational normal form (TGFSR(R)), with state bit reflection and tempering. The basic idea is to define a series x_i through a simple recurrence relation, and then output numbers of the form $x_i T$, where T is an invertible F_2 matrix called a tempering matrix [1].

Compared to LCG, MT has a very long period length (2^{19937}), which is enough for nearly any use of random numbers. Plus, it is easy to be modified to become cryptographically capable, with efficiency similar to LCG. Thus, it is now most popular RNG algorithm. In this article, we will use *MT19937*, a modified version of MT as our method to generate uniform random variables. Since it is not our focus to generate random numbers, the detailed MT algorithm is omitted and the C++ code, *twister.cpp* to implement the algorithm is provided by the authors of the algorithm [1], as included in Appendix.

It's noteworthy that a integer seed must be provided for *twister* every time when the MATLAB starts. It's suggested to a random seed like the current time. Thus we run `twist('state', 100*sum(clock))` to initialize the generator.

A sample of 50 random uniform variables in $[0, 1]$ by calling *twister(1, 50)* is listed in [Table I](#).

TABLE I
A SAMPLE OF 50 UNIFORM RANDOM VARIABLES

0.557623	0.096574	0.728721	0.946501	0.548659	0.410843	0.017287	0.627216	0.230289	0.896297
0.224381	0.528917	0.042601	0.202466	0.696846	0.382712	0.464360	0.399971	0.817837	0.768933
0.887287	0.305046	0.486443	0.392051	0.423167	0.634413	0.174765	0.631697	0.150344	0.422200
0.806684	0.324624	0.197380	0.765630	0.732800	0.935835	0.302143	0.493050	0.181880	0.654789
0.823832	0.141927	0.486791	0.435605	0.864884	0.457860	0.747067	0.605341	0.627621	0.112581

III. BINOMIAL RANDOM VARIABLES

A. General Transformation Method

The algorithm of general transformation method to generate any required type of random variables given their PDF or PMF is as follows:

- 1) Generate a uniformly distributed random number u in the interval $[0, 1]$.
- 2) For random variable X with CDF $F(x)$, find x such that $F(x) = u$ or $x = F^{-1}(u)$. Then x is the required random variable specified by $F(x)$.

Note that since the close form of $F^{-1}(u)$ is in most cases, not readily available, we adopt a numeric method, that is, finding the minimum x such that $F(x) \geq u$. Also, we don't necessarily need the close form for $F(x)$ as we note $F(x) = \sum_{y=x_i}^x f(y)$

for discrete random variables and $F(x) = \int_{-\infty}^x f(y)dy$.

B. Generation

The PMF of a binomial random X with parameter n and p is

$$f(x) = C_n^x p^x (1-p)^{n-x}, 0 \leq x \leq n, x \in N^* \quad (2)$$

Note that the close form of the PMF is not readily available, we apply

$$F(x) = \begin{cases} \sum_{y=0}^{\lfloor x \rfloor} f(y), & 0 \leq x \leq n \\ 1, & x > n \\ 0, & x < 0 \end{cases} \quad (3)$$

for evaluation.

BinomialGenerator.m illustrates the approach of generating any given number N of Binomial random variables with probability of success p and number of trials n according to the transformation method. A sample of $N = 50X's$ with $n = 100$, $p = 0.6$ is in Table II. During the generation process of large N (for example $N = 1000$), the cost of calculating C_n^x

TABLE II
A SAMPLE OF 50 BINOMIAL RANDOM VARIABLES

58	55	54	56	66	54	60	64	54	54	62	59	49	67	64	51	59	65	68	55	58	68	64	58	60
57	66	57	56	53	61	65	54	65	52	60	53	62	62	62	59	60	61	62	51	58	54	63	54	62

is overwhelmingly expensive. In our environment, it can take nearly half an hour (1703.360s) to finish the calculation and thus obviously unacceptable. In addition, calculating C_n^x for large n leads to loss of accuracy since this coefficient can be much larger than 9×10^{15} and in *MATLAB* it is reduced to only the first 15 digits.

Hence, we consider three alternatives to solve the problem.

1) *Generate from Bernoulli Random Variables*: Since binomial random variable is defined as the number of successes of a series of Bernoulli trials, we could sum Bernoulli random variables up to obtain binomial random variables, that is,

$$X = \sum_{i=1}^n Y_i \quad (4)$$

where X denotes a binomial random variables with parameters n and p , and Y_i denotes a mutually independent Bernoulli random variables with probability of success p , whose PMF and CDF are

$$f(y) = \begin{cases} p, & y = 1 \\ 1 - p, & y = 0 \end{cases} \quad (5)$$

and

$$F(y) = \begin{cases} 0, & y < 0 \\ 1 - p, & 0 \leq y < 1 \\ 1, & y \geq 1 \end{cases} \quad (6)$$

Thus, we could first use transformation method to generate Bernoulli random variables with the same p as required for binomial random variables and then count the number of success among n independent Bernoulli random variables, which leads to the required binomial random variable. *BernoulliGenerator.m* illustrates the approach of generating any given number n of Bernoulli random variables with probability of success p according to the transformation method and based on that, *CountSuccessBinomialGenerator.m* shows the process of generating binomial random variables in this way.

In our environment, the process of generating a $N = 1000$ sample lasts only for seconds (1.744s), thus is indeed an acceptable method.

A further examination of the built-in generator *binornd.m* indicates that it adopts the same method, that is, generating values as a sum of Bernoulli random variables. The efficiency and other issues of this method is discussed in detail in [2].

2) *Approximate the CDF with normal PDF*: Note that $np = 60 > 5$ and $n(1 - p) = 40 > 5$ in our case, the normal approximation to the binomial distribution is guaranteed to be of good precision. Thus, with continuity correction applied, we have

$$F(x) = P(X \leq x) = P(Z \leq \frac{x + 0.5 - np}{\sqrt{np(1 - p)}}) \quad (7)$$

This and the next methods are implemented in *BinomialGeneratorDiscussion.m* with $mode = 1, 2$ respectively. This method usually requires half a minute (28.631s) to finish.

3) *Directly use built-in functions for calculating binomial PMF*: The built-in solution for PMF of binomial distribution $f(x)$ is *binopdf.m* which is based on a saddle point expansion [3]:

$$\log(f(x; n, p)) = \log(f(x; n, \frac{x}{n})) - D(x; n, p) \quad (8)$$

where the deviance $D(x; n, p)$ is defined as

$$D(x; n, p) = \log(f(x; n, \frac{x}{n})) - \log(f(x; n, p)) = x \log(\frac{x}{np}) + (n - x) \log(\frac{n - x}{n(1 - p)}) \quad (9)$$

A generation method using the built-in binomial PMF requires several minutes (294.399s) to finish.

C. Verification

For the rest part of this problem, for efficiency and accuracy, all random variables are generated through the approach in [Subsubsection III-B1](#) instead of the general transformation method in [Subsection III-B](#).

1) *Numerical Characteristics*: Execute *CountingSuccessBinomialGenerator.m* with $N = 1000$, $n = 100$ and $p = 0.6$, we generate a sample of 1000 binomial random variables X with theoretic mean

$$\mu = np = 60 \quad (10)$$

and variance

$$\sigma^2 = np(1 - p) = 24. \quad (11)$$

Passing X to *MeanVariance.m*, we have

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n} = 60.0460, s^2 = \frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1} = 24.7787 \quad (12)$$

which are consistent with [Equation 10](#) and [Equation 11](#).

2) *Normalized Histogram*: The normalized histogram of the sample is a good approximation to the PMF of the population as long as the size of the sample is fairly big enough. The drawing process is accomplished by *DrawingHistogram1.m*. The second parameter *width* is to determine the width for each bin. The different choice of *width* may have critical influence on the shape of the histogram. Apart from the normalized histogram, this program also draws the theoretic PMF of PDF of that distribution for comparison.

Running *DrawingHistogram.m* with generated X , $width = 1$, we have the output figure as [Figure 1](#).

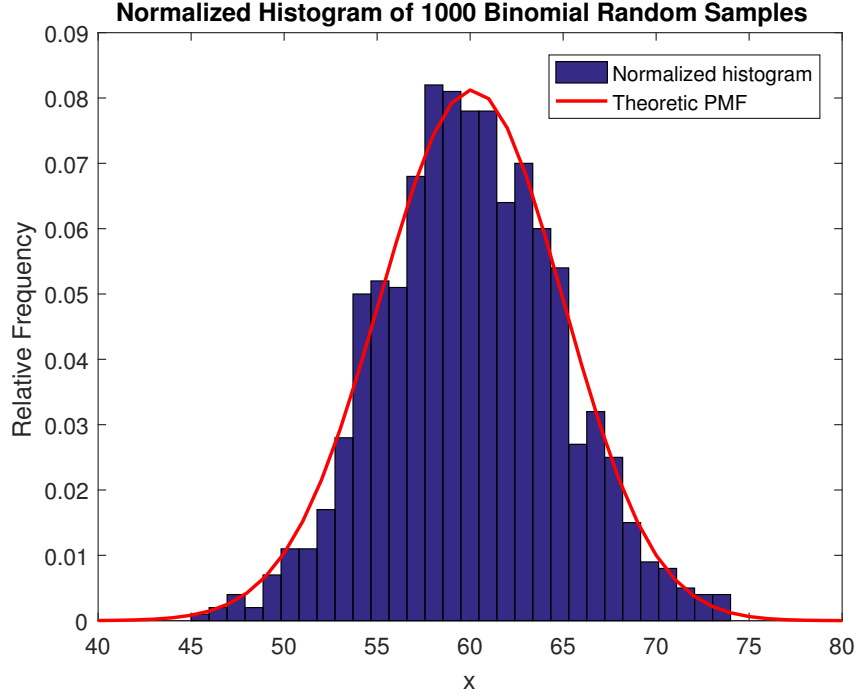


Fig. 1. Normalized Histogram for 1000 Binomial Random Samples.

3) *Common Verification Fallacies*: In Figure 1, it is clear that the sample approximately follows a binomial distribution. However, we still need statistical methods to verify it. Students turn to use covariance or correlation coefficient between samples generated from standard generator and their generator to evaluate whether they succeed or not, which is a complete mistake theoretically and practically.

The first common fallacy is to directly compute correlation coefficient between two generated samples.

Theoretically speaking, a generator always generates *independent* random samples from a given distribution. After all, it can't be called a *random* variable generator if the present sample is related to previous ones. Thus, even two samples X_1, X_2 of N random variables generated from the standard generator are independent and thus uncorrelated, that is

$$\lim_{N \rightarrow \infty} \text{cov}(X_1, X_2) = 0 \quad (13)$$

In practice, *ExplainingCommonFallacy.m* computes correlation coefficients between various generated samples to indicate these common fallacies. Run that with *mode* = 1, we compute the correlation coefficient between two sample sets. Both are generated from standard binomial random variable generator *binornd.m*, with $N = 1000$, $n = 100$ and $p = 0.6$. The return value is

$$\rho = \frac{\text{cov}(X_1, X_2)}{s_1 s_2} = -0.0066 \approx 0 \quad (14)$$

As is obvious in Equation 14, there is no correlation between samples generated from the same standard generator.

The second common fallacy is to compute correlation coefficient between two sorted generated samples (say, in an increasing order).

This method seems swifter than the first one but still lacks both theoretical background and practical meanings. In fact, when sorted, the samples become *first order statistics*. And respective PMF becomes [4]

$$f_k(x_k) = \frac{n!}{(k-1)!(n-k)!} [F(x_k)]^{k-1} [1 - F(x_k)]^{n-k} f(x_k) \quad (15)$$

where x_k is the k th order statistic. Therefore, the theoretic covariance is computed by

$$\text{cov}(X_1, X_2) = E\left(\frac{\sum_{k=0}^N (x_{1k} - \bar{x}_1)(x_{2k} - \bar{x}_2)}{s_{x1} s_{x2}}\right) \quad (16)$$

where x_{1k} and x_{2k} are k th order sample from X_1 and X_2 and follow distribution from Equation 15. Theoretic value of Equation 16 is hard to evaluate but what we need to know is that it has nothing to do with the validity of the generator.

Run *ExplainingCommonFallacy.m* with $mode = 2$, we compute the the correlation coefficient between two binomial sample sets with $N_1 = N_2 = 1000$, $n_1 = n_2 = 100$, $p_1 = 0.1$, $p_2 = 0.9$ respectively. The return value is

$$\rho = \frac{cov(X_1, X_2)}{s_1 s_2} = 0.9800 \quad (17)$$

We see that even if the two distribution is different to a very large extent, the correlation coefficient still approaches 1.

D. Non-parametric Hypothesis Test

A systematic approach to test whether the samples fit a given distribution is Pearson's χ^2 test. It tests a null hypothesis H_0 stating that the frequency distribution of certain events observed in a sample is consistent with a particular theoretical distribution. The test statistic is

$$\chi^2 = \sum_{i=1}^n \frac{(O_i - E_i)^2}{E_i} = N \sum_{i=1}^n \frac{(O_i/N - p_i)}{p_i} \quad (18)$$

where O_i is the number of observations in category i , $E_i = Np_i$ is the expected frequency of type i , asserted by the null hypothesis that the fraction of type i in the population is p_i and N is the total number of observations (sample size). Then the test statistic follows a χ^2 distribution with degree of freedom $n - 1 - p$, where p is the number of the unknown parameters of the distribution if H_0 is true, which would be replaced by their maximum likelihood estimates in the calculation process.

NonParametricHypothesisTest.m implements all the non-parameter tests for this and following problem. Here, we test whether our generated sample X fits the theoretical distribution. Run *NonParametricHypothesisTest.m* with X and $mode = 1$, we have our hypothesis result and respective p value.

$$\text{Accept } H_0, p = 0.8858 \quad (19)$$

Thus, we can't reject H_0 , which indicates the frequency distribution from our samples is consistent with the theoretical one.

IV. TWO INDEPENDENT NORMAL RANDOM VARIABLES

A. Generation

Here, we apply a special form of transformation method to generate two independent normal random variables. As long as X_1 and X_2 are two independent uniform random variables,

$$Y_1 = \sigma \sqrt{-2 \ln X_1} \cos(2\pi X_2) + m \quad (20)$$

$$Y_2 = \sigma \sqrt{-2 \ln X_1} \sin(2\pi X_2) + m \quad (21)$$

are two independent normal random variables with $E(Y_1) = E(Y_2) = m$, $V(Y_1) = V(Y_2) = \sigma^2$.

NormalGenerator.m illustrates the approach of generating N samples of two independent normal random variable with parameter m and σ . Table III and Table IV records two sets of 50 normal random samples Y_1 and Y_2 with $m = 1$ and $\sigma = 2$.

TABLE III
A SAMPLE OF 50 NORMAL RANDOM VARIABLES, FIRST SET

1.84735	-1.04541	1.28496	1.00590	-0.75245	1.32464	0.93434	-1.03333	0.15877	-0.14319
2.81723	-0.09764	2.28270	2.06963	-2.96432	-1.08785	0.30621	3.78081	3.77749	3.15242
0.59305	0.53039	1.96144	2.89554	1.48787	2.92631	-2.66943	1.17166	1.82833	0.27576
2.73823	3.92442	-0.11542	2.39502	-1.64079	2.18286	1.17879	0.87407	-0.52520	1.68041
-0.75731	1.14655	-2.55063	-2.64490	-1.03768	1.43336	-1.29162	2.69257	2.95180	-0.43551

TABLE IV
A SAMPLE OF 50 NORMAL RANDOM VARIABLES, SECOND SET

-0.45481	-0.80005	1.90802	1.55385	1.28825	2.48323	0.94621	2.11215	1.70294	3.24939
-1.67786	1.68190	2.15890	0.21632	1.29920	-0.26320	-0.17812	-2.26349	1.42435	1.21952
-0.44189	-1.72672	0.02997	0.28586	1.95675	-0.87463	-0.40302	0.78905	-0.15996	-3.33380
-0.53905	3.89746	-0.87898	0.99720	-1.90732	-1.33349	-1.74078	-0.17804	-0.87675	0.41310
2.86881	0.90354	3.31284	3.19603	0.33660	-0.63267	3.04379	1.29074	1.78196	2.45101

B. Verification

1) *Numerical Characteristics*: Execute *NormalGenerator.m* with $N = 1000$, $m = 1$ and $\sigma = 2$, we generate two sets of 1000 binomial random samples Y_1 and Y_2 with theoretic mean and variance $E(Y_1) = E(Y_2) = 1$ and $V(Y_1) = V(Y_2) = 4$.

Passing Y_1 and Y_2 to *MeanVariance.m*, we have

$$\bar{y}_1 = 1.1198, \bar{y}_2 = 0.9554 \quad (22)$$

$$s_1^2 = 4.0348, s_2^2 = 3.7386 \quad (23)$$

which are consistent with the theoretical values.

In addition, we compute their correlation coefficient by calling *corrcoef(Y1,Y2)* and get

$$\rho = 0.0046 \quad (24)$$

which shows that Y_1 and Y_2 is approximately uncorrelated.

C. Normalized Histogram

Run *HistogramDrawing2.m* with Y_1 , Y_2 and *width* = 0.5, we have the histograms as well as the theoretic PDF in Figure 2.

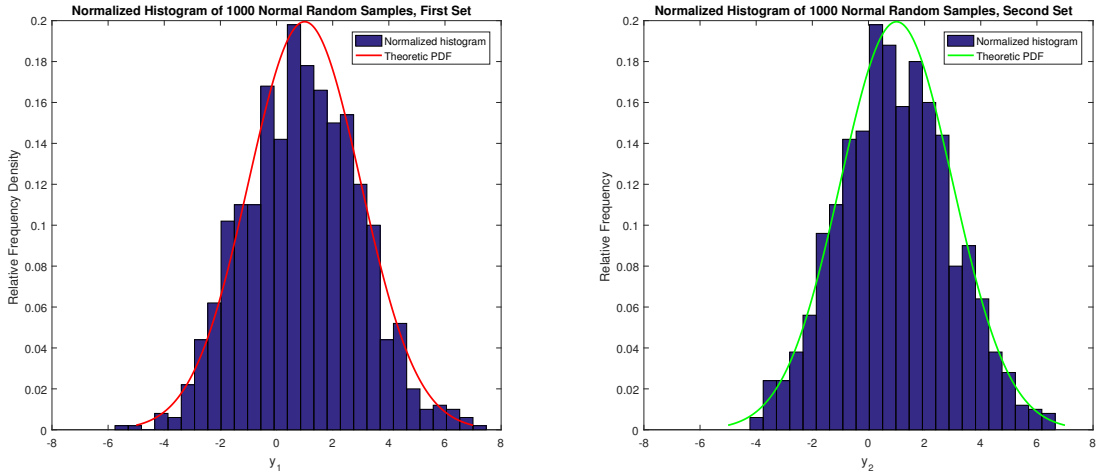


Fig. 2. Normalized Histogram for Two Sets of 1000 Random Samples.

D. Verification

1) *Test of Normality*: Again, we use Pearson's χ^2 test to check whether these two sample sets follows the theoretical normal distribution. Run *NonParametricHypothesisTest.m* with previous Y_1 and Y_2 with *mode* = 2, we have

$$\text{Accept } H_0, p = 0.8107 \text{ for } Y_1 \quad (25)$$

$$\text{Accept } H_0, p = 0.8009 \text{ for } Y_2 \quad (26)$$

Thus, we can't reject H_0 and conclude that Y_1 and Y_2 follow the theoretical distribution.

2) *Test of Independence*: The systematic approach to test independence between two samples is another important application of Pearson's χ^2 test. In that case, the two samples are considered together as a sample of 2-tuples, where the first dimension is consisted of r categories and the second s . Then it tests a null hypothesis H_0 stating that the frequency of observations falling into i th category of the first dimension and j th category of the second is exactly the same as the frequency of i th category of the first multiplies that of j th category of the second, that is

$$H_0 : P(Y_1 \in \text{Category}_i, Y_2 \in \text{Category}_j) = P(Y_1 \in \text{Category}_i)P(Y_2 \in \text{Category}_j) \quad (27)$$

Then, the constructed test statistic is

$$\chi^2 = \sum_{i=1}^r \sum_{j=1}^s \left(n_{ij} - \frac{n_{i*}n_{*j}}{n} \right)^2 / \frac{n_{i*}n_{*j}}{n} \quad (28)$$

where n_{ij} is the number of observations falling into both i th category of the first dimension and j th category of the second, n_{i*} , n_{*j} is the number of observations falling into i th category of the first dimension and j th category of the second dimension

respectively and n is the total number of observations. Then the test statistic follows a χ^2 distribution with degree of freedom $(r-1)(s-1)$.

In our test, we first divide the two sets of normal random samples into 6 categories each according to their values, that is

$$(-\infty, m-2\sigma), [m-2\sigma, m-\sigma), [m-\sigma, m), [m, m+\sigma), [m+\sigma, m+2\sigma), [m+2\sigma, +\infty)$$

Run *IndependenceTest.m* with generated Y_1 and Y_2 , we have the generated contingency table as Table V and the test result is

$$\text{Accept } H_0, p = 0.5588 \quad (29)$$

TABLE V
CONTINGENCY TABLE FOR TEST OF INDEPENDENCE BETWEEN Y_1 AND Y_2

$Y_2 \backslash Y_1$	$(-\infty, m-2\sigma]$	$[m-2\sigma, m-\sigma)$	$[m-\sigma, m)$	$[m, m+\sigma)$	$[m+\sigma, m+2\sigma)$	$[m+2\sigma, +\infty)$	Total
$(-\infty, m-2\sigma]$	1	3	5	4	1	0	14
$[m-2\sigma, m-\sigma)$	0	21	61	46	29	5	162
$[m-\sigma, m)$	5	45	110	118	43	8	329
$[m, m+\sigma)$	12	41	120	111	40	9	333
$[m+\sigma, m+2\sigma)$	3	21	48	47	20	0	139
$[m+2\sigma, +\infty)$	1	1	10	8	3	0	23
Total	22	132	354	334	136	22	1000

Thus, we can't reject H_0 and conclude that Y_1 and Y_2 are mutually independent.

V. CONCLUSION

In this article, we introduce methods for generating pseudo-random numbers, generate typical random variables using transformation method based on them, and apply statistical hypothesis tests to verify that they have satisfied the requirements. In addition, we extend our discussion to the efficiency and accuracy of different generation methods, analyze the algorithms of *MATLAB* built-in random variable generators and also explaining some common fallacies when verifying the generated samples follow a certain distribution. The whole project is insightful and can serve as the first step for newcomers in the field of probability and signal analysis.

ACKNOWLEDGEMENT

The author would like to express his gratitude to Dr. S.C. Wu in National Tsing Hua University, who briefly introduced the transformation method and supervised part of this project, and Dr Y. Song in University of Electronic Science and Technology of China, under whose supervision and guidance the most part of this project was conducted.

REFERENCES

- [1] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.
- [2] L. Devroye, "Sample-based non-uniform random variate generation," in *Proceedings of the 18th conference on Winter simulation*. ACM, 1986, pp. 260–265.
- [3] C. Loader, "Fast and accurate computation of binomial probabilities," 2000.
- [4] H. A. David and H. N. Nagaraja, *Order statistics*. Wiley Online Library, 1981.

APPENDIX

The source codes are listed here for reference.

1) *BernoulliGenerator.m*

```

1  %{
2  %Generating n Bernoulli random variables with probability of success p.
3  %Zexi Huang
4  %Oct. 4 2016
5  %}
6
7  function Y=BernoulliGenerator(n,p)
8  %n: number of output random variables.
9  %p: probability of success.
10
11 %CDF of Bernoulli RV.
12 function F=CDF(x)
```

```

13     if x<0
14         F=0;
15     elseif x>=1
16         F=1;
17     else
18         F=1-p;
19     end
20 end
21
22 %Numerical approach of the inverse CDF.
23 function G=InverseCDF(u)
24     x=[0,1];
25     for s=x
26         if CDF(s)>=u
27             G=s;
28             break;
29         end
30     end
31 end
32
33 %Generation process
34 for ii=n:-1:1
35     %Seemingly more efficient than from 1 to n. Array size only changed
36     %once in the loop.
37     u=twister();
38     Y(ii)=InverseCDF(u);
39 end
40
41 end

```

2) BinomialGenerator.m

```

1  %{
2  %Generating N binomial random variables with parameter p and n.
3  %Zexi Huang
4  %Oct. 4 2016
5  %}
6
7  function X=BinomialGenerator(N,n,p)
8  %N: number of output random variables.
9  %n: number of trials.
10 %p: probability of success.
11
12 %PMF of binomial RV.
13 function f=PMF(x)
14     f=nchoosek(n,x)*p^x*(1-p)^(n-x);
15 end
16
17 %CDF of binomial RV (evaluation).
18 function F=CDF(x)
19     if x<0
20         F=0;
21     elseif x>n
22         F=1;
23     else
24         F=0;
25         for ii=1:floor(x)
26             F=F+PMF(ii);
27         end
28     end
29 end
30
31 %Numerical approach of the inverse CDF.
32 function G=InverseCDF(u)
33     x=0:n;
34     for s=x
35         if CDF(s)>u
36             G=s;
37             break;
38         end
39     end
40 end
41
42 %Generation process
43 %Seemingly more efficient than from 1 to n. Array size only changed
44 %once in the loop.
45 for jj=N:-1:1

```



```

46     u=twister();
47     X(jj)=InverseCDF(u);
48 end
49
50 end

```

3) *BinomialGeneratorDiscussion.m*

```

1  %{
2  %Generating N Binomial random variables with parameter p and n with two
3  %alternative approaches.
4  %Zexi Huang
5  %Oct. 4 2016
6  %}
7  function X=BinomialGeneratorDiscussion(N,n,p,mode)
8  %N: number of output random variables.
9  %n: number of trials.
10 %p: probability of success.
11 %mode=1: Approximate with normal PDF.
12 %mode=2: Use built-in binopdf.
13
14 %PMF of binomial RV.
15 function f=PMF(x)
16     switch mode
17         case {1}
18             f=normpdf(x,n*p,sqrt(n*p*(1-p)));
19         case {2}
20             f=binopdf(x,n,p);
21     end
22 end
23
24 %CDF of binomial RV (evaluation).
25 function F=CDF(x)
26     if (x<0)
27         F=0;
28     elseif x>n
29         F=1;
30     else
31         F=0;
32         for ii=1:floor(x)
33             F=F+PMF(ii);
34         end
35     end
36 end
37
38 %Numerical approach of the inverse CDF.
39 function G=InverseCDF(u)
40     x=0:n;
41     for s=x
42         if CDF(s)>u
43             G=s;
44             break;
45         end
46     end
47 end
48
49
50 %Generation process
51 %Seemingly more efficient than from 1 to n. Array size only changed
52 %once in the loop.
53 for jj=N:-1:1
54     u=twister();
55     X(jj)=InverseCDF(u);
56 end
57
58 end

```

4) *CountingSuccessBinomialGenerator.m*

```

1  %{
2  %Generating N Binomial random variables with parameter p and n based on
3  %generated independent Bernoulli random variables with same parameter p.
4  %Zexi Huang
5  %Oct. 4 2016
6  %}
7
8  function X=CountingSuccessBinomialGenerator(N,n,p)
9  %N: number of output random variables.

```

```

10 %n: number of Bernoulli random variables to be counted.
11 %p: probability of success.
12
13 %Counting number of success in y.
14 function count=CountSuccess(y)
15     count=0;
16     for x=y
17         if x==1
18             count=count+1;
19         end
20     end
21 end
22
23 %Generation process
24
25 for ii=N:-1:1
26     y=BernoulliGenerator(n,p);
27     X(ii)=CountSuccess(y);
28 end
29
30 end

```

5) *DrawingHistogram1.m*

```

1  %{
2  %Drawing normalized histogram derived from a sample of binomial distribution
3  %and compare it with theoretic PMF.
4  %Zexi Huang
5  %Oct. 5 2016
6  %}
7
8  function DrawingHistogram1(X, width)
9  %X: sample of random variables to be drawn.
10 %width: width of each bin.
11
12
13 %Produce histogram.
14 [nelements, xcenters]=hist(X,(range(X)+1)/width);
15 bar(xcenters,nelements/(width*length(X)),1);
16 xlabel('x');
17 ylabel('Relative Frequency');
18 hold on;
19
20 %Produce respective figure.
21 plot(40:80,binopdf(40:80,100,0.6),'r','LineWidth',1.5);
22 legend('Normalized histogram','Theoretic PMF');
23 title('Normalized Histogram of 1000 Binomial Random Samples');
24
25 hold off;
26
27
28 end

```

6) *DrawingHistogram2.m*

```

1  %{
2  %Drawing normalized histogram derived from two samples of normal distribution
3  %and compare them with their theoretic PMF.
4  %Zexi Huang
5  %Oct. 5 2016
6  %}
7
8  function DrawingHistogram2(Y1,Y2,width)
9  %Y1,Y2: sample of random variables to be drawn.
10 %width: width of each bin.
11
12 %Produce figure for Y1.
13 subplot(1,2,1);
14 [nelements, xcenters]=hist(Y1,(range(Y1)+1)/width);
15 bar(xcenters,nelements/(width*length(Y1)),1);
16 xlabel('y_1');
17 ylabel('Relative Frequency Density');
18 hold on;
19 plot(-5:0.01:7,normpdf(-5:0.01:7,1,2),'r','LineWidth',1.5);
20 legend('Normalized histogram','Theoretic PDF');
21 title('Normalized Histogram of 1000 Normal Random Samples, First Set');
22 %set(gca,'XTick',-10:1:10);
23 axis([-8 8 0 0.2]);

```

```

24
25
26 %Produce figure for Y2.
27 subplot(1,2,2);
28 [nelements , xcenters]=hist(Y2,(range(Y2)+1)/width);
29 bar(xcenters ,nelements/(width*length(Y2)),1);
30 xlabel('y_2');
31 ylabel('Relative Frequency');
32 hold on;
33 plot(-5:0.01:7,normpdf(-5:0.01:7,1,2),'g','LineWidth',1.5);
34 legend('Normalized histogram','Theoretic PDF');
35 title('Normalized Histogram of 1000 Normal Random Samples, Second Set');
36 %set(gca,'XTick',-10:1:10);
37 axis([-8 8 0 0.2]);
38
39 hold off;
40
41
42 end

```

7) ExplainingCommonFallacy.m

```

1 %{
2 %Explain common fallacy in practice.
3 %Zexi Huang
4 %Oct. 5 2016
5 %}
6 function coef=ExplainingCommonFallacy(mode)
7 %mode: determines which set of coefficient to return.
8
9
10 %Generating samples of RV and compute their correlation coefficient.
11 switch mode
12     case {1}
13         X1=binornd(100,0.6,1000,1);
14         X2=binornd(100,0.6,1000,1);
15
16     case {2}
17         % x=1*ones(5000000,1);
18         % X1=chi2rnd(x);
19         % X1=sort(X1);
20         % y=zeros(5000000,1);
21         % X2=normrnd(y,x);
22         % X2=sort(X2);
23         X1=binornd(100,0.1,1000,1);
24         X1=sort(X1);
25         X2=binornd(100,0.9,1000,1);
26         X2=sort(X2);
27     end
28
29 coef=corrcoef(X1,X2);
30 coef=coef(1,2);
31 end

```

8) IndependenceTest.m

```

1 %{
2 %Independent test between two normal random samples.
3 %Zexi Huang
4 %Oct. 6 2016
5 %}
6
7 function [table ,h,p]=IndependenceTest(Y1,Y2)
8 %Y1, Y2: two sets random samples to be tested.
9 %table: the output contingency table.
10 %h: whether the test is rejected, 1 indicates rejected.
11 %p: p-value of the test.
12
13 %Computing mean, sample standard deviation and lengths of Y1, Y2
14 [m1,v1]=MeanVariance(Y1);
15 [m2,v2]=MeanVariance(Y1);
16 s1=sqrt(v1);
17 s2=sqrt(v2);
18 n1=numel(Y1);
19 n2=numel(Y2);
20
21 %Replacing values with category labels.
22 for ii=1:n1

```

```

23     if (Y1(ii) < m1 - 2*s1)
24         Y1(ii) = 1;
25     elseif (Y1(ii) < m1 - s1)
26         Y1(ii) = 2;
27     elseif (Y1(ii) < m1)
28         Y1(ii) = 3;
29     elseif (Y1(ii) < m1 + s1)
30         Y1(ii) = 4;
31     elseif (Y1(ii) < m1 + 2*s1)
32         Y1(ii) = 5;
33     else
34         Y1(ii) = 6;
35     end
36 end
37
38 for ii = 1:n2
39     if (Y2(ii) < m2 - 2*s2)
40         Y2(ii) = 1;
41     elseif (Y2(ii) < m2 - s2)
42         Y2(ii) = 2;
43     elseif (Y2(ii) < m2)
44         Y2(ii) = 3;
45     elseif (Y2(ii) < m2 + s2)
46         Y2(ii) = 4;
47     elseif (Y2(ii) < m2 + 2*s2)
48         Y2(ii) = 5;
49     else
50         Y2(ii) = 6;
51     end
52 end
53
54
55 %Produce the contingency table and do the chi2 test.
56 [table, chi2, p] = crosstab(Y1, Y2);
57
58 if (p < 0.05)
59     h = 1;
60 else
61     h = 0;
62 end
63
64 end

```

9) MeanVariance.m

```

1  %{
2  %Calculating mean and variance for given random samples.
3  %Zexi Huang
4  %Oct. 5 2016
5  %}
6  function [mean, variance] = MeanVariance(X)
7  %X: an array of random samples.
8  %mean, variance: sample mean and sample variance.
9
10 %Size of sample.
11 n = numel(X);
12
13 %Mean of sample.
14 mean = sum(X) / n;
15
16 %Variance of sample.
17 variance = 0;
18 for jj = 1:n
19     variance = variance + (X(jj) - mean)^2;
20 end
21 variance = variance / (n - 1);
22
23 end

```

10) NonParametricHypothesisTest.m

```

1  %{
2  %Non-parameter hypothesis test for good of fit and independence.
3  %Zexi Huang
4  %Oct. 5 2016
5  %}
6
7  function [h, p] = NonParametricHypothesisTest(X, mode)

```

```

8 %X: sample of random variables to be tested.
9 %mode: determines which test is used.
10 %h: whether the test is rejected, 1 indicates rejected.
11 %p: p-value of the test.
12
13 %Generate standard pdf.
14 switch mode
15     case{1}
16         pd=makedist('Binomial','N',100,'p',0.6);
17         [h,p]=chi2gof(X,'CDF',pd,'Ctrs',[45,50,55,60,65,70,75]);
18     case{2}
19         pd=makedist('Normal','mu',1,'sigma',2);
20         [h,p]=chi2gof(X,'CDF',pd);
21
22 end
23
24 %Hypothesis test.

```

11) NormalGenerator.m

```

1 %{
2 %Generating N normal random variables with given mean and standard
3 %deviation.
4 %Zexi Huang
5 %Oct. 5 2016
6 %}
7
8 function [Y1,Y2]=NormalGenerator(N,mean,sd)
9 %N: number of random variables to be generated.
10 %mean: mean of required normal distribution.
11 %sd: standard deviation of required normal distribution.
12
13 %Generate uniform random variables sequence.
14 X1=twister(N,1);
15 X2=twister(N,1);
16
17 %Generate normal random variables.
18 Y1=sd*sqrt(-2*log(X1)).*cos(2*pi*X2)+mean;
19 Y2=sd*sqrt(-2*log(X1)).*sin(2*pi*X2)+mean;
20
21 end

```

12) twister.cpp

```

1 #include "mex.h"
2 #include "matrix.h"
3 #include <math.h>
4 #include <string.h>
5 #include <ctype.h>
6
7
8 /* *****
9 *
10 * Mersenne Twister code:
11 */
12
13
14 /*
15  A C-program for MT19937, with initialization improved 2002/1/26.
16  Coded by Takuji Nishimura and Makoto Matsumoto.
17
18  Before using, initialize the state by using init_genrand(seed)
19  or init_by_array(init_key, key_length).
20
21  Copyright (C) 1997 – 2002, Makoto Matsumoto and Takuji Nishimura,
22  All rights reserved.
23
24  Redistribution and use in source and binary forms, with or without
25  modification, are permitted provided that the following conditions
26  are met:
27
28  1. Redistributions of source code must retain the above copyright
29     notice, this list of conditions and the following disclaimer.
30
31  2. Redistributions in binary form must reproduce the above copyright
32     notice, this list of conditions and the following disclaimer in the
33     documentation and/or other materials provided with the distribution.
34

```

```

35     3. The names of its contributors may not be used to endorse or promote
36     products derived from this software without specific prior written
37     permission.
38
39     THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
40     "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
41     LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
42     A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
43     CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
44     EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
45     PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
46     PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
47     LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
48     NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
49     SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
50
51
52     Any feedback is very welcome.
53     http://www.math.keio.ac.jp/matsumoto/emt.html
54     email: matsumoto@math.keio.ac.jp
55 */
56
57 /* Period parameters */
58 #define N 624
59 #define M 397
60 #define MATRIX_A 0x9908b0dfUL /* constant vector a */
61 #define UPPER_MASK 0x80000000UL /* most significant w-r bits */
62 #define LOWER_MASK 0x7fffffffUL /* least significant r bits */
63
64 static unsigned long mt[N]; /* the array for the state vector */
65 static int mti=N+1; /* mti==N+1 means mt[N] is not initialized */
66
67 /* initializes mt[N] with a seed */
68 void init_genrand(unsigned long s)
69 {
70     mt[0]= s & 0xffffffffUL;
71     for (mti=1; mti<N; mti++) {
72         mt[mti] =
73             (1812433253UL * (mt[mti-1] ^ (mt[mti-1] >> 30)) + mti);
74         /* See Knuth TAOCP Vol2. 3rd Ed. P.106 for multiplier. */
75         /* In the previous versions, MSBs of the seed affect */
76         /* only MSBs of the array mt[]. */
77         /* 2002/01/09 modified by Makoto Matsumoto */
78         mt[mti] &= 0xffffffffUL;
79         /* for >32 bit machines */
80     }
81 }
82
83 /* initialize by an array with array-length */
84 /* init_key is the array for initializing keys */
85 /* key_length is its length */
86 void init_by_array(unsigned long init_key[], unsigned long key_length)
87 {
88     int i, j, k;
89     init_genrand(19650218UL);
90     i=1; j=0;
91     k = (N>key_length ? N : key_length);
92     for (; k; k--) {
93         mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1664525UL))
94             + init_key[j] + j; /* non linear */
95         mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
96         i++; j++;
97         if (i>=N) { mt[0] = mt[N-1]; i=1; }
98         if (j>=key_length) j=0;
99     }
100     for (k=N-1; k; k--) {
101         mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1566083941UL))
102             - i; /* non linear */
103         mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
104         i++;
105         if (i>=N) { mt[0] = mt[N-1]; i=1; }
106     }
107
108     mt[0] = 0x80000000UL; /* MSB is 1; assuring non-zero initial array */
109 }
110
111 /* generates a random number on [0,0xffffffff]-interval */

```

```

112 unsigned long genrand_int32(void)
113 {
114     unsigned long y;
115     static unsigned long mag01[2]={0x0UL, MATRIX_A};
116     /* mag01[x] = x * MATRIX_A for x=0,1 */
117
118     if (mti >= N) { /* generate N words at one time */
119         int kk;
120
121         if (mti == N+1) /* if init_genrand() has not been called, */
122             init_genrand(5489UL); /* a default initial seed is used */
123
124         for (kk=0;kk<N-M;kk++) {
125             y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
126             mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1UL];
127         }
128         for (;kk<N-1;kk++) {
129             y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
130             mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1UL];
131         }
132         y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
133         mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1UL];
134
135         mti = 0;
136     }
137
138     y = mt[mti++];
139
140     /* Tempering */
141     y ^= (y >> 11);
142     y ^= (y << 7) & 0x9d2c5680UL;
143     y ^= (y << 15) & 0xefc60000UL;
144     y ^= (y >> 18);
145
146     return y;
147 }
148
149 /* generates a random number on [0,0x7fffffff]—interval */
150 long genrand_int31(void)
151 {
152     return (long)(genrand_int32()>>1);
153 }
154
155 /* generates a random number on [0,1]—real—interval */
156 double genrand_real1(void)
157 {
158     return genrand_int32()*(1.0/4294967295.0);
159     /* divided by 2^32-1 */
160 }
161
162 /* generates a random number on [0,1)–real–interval */
163 double genrand_real2(void)
164 {
165     return genrand_int32()*(1.0/4294967296.0);
166     /* divided by 2^32 */
167 }
168
169 /* generates a random number on (0,1)–real–interval */
170 double genrand_real3(void)
171 {
172     return (((double)genrand_int32()) + 0.5)*(1.0/4294967296.0);
173     /* divided by 2^32 */
174 }
175
176 /* generates a random number on [0,1) with 53-bit resolution */
177 double genrand_res53(void)
178 {
179     unsigned long a=genrand_int32()>>5, b=genrand_int32()>>6;
180     return (a*67108864.0+b)*(1.0/9007199254740992.0);
181 }
182 /* These real versions are due to Isaku Wada, 2002/01/09 added */
183
184
185 /* *****
186 *
187 * MATLAB code:
188 */

```

```

189
190 /* the gateway function */
191 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
192 {
193     int errMsgNum = 0;
194
195     /* First time through.
196     */
197     if (mti == N+1) init_genrand(5489UL);
198
199     if (nlhs > 1) {
200         mexErrMsgIdAndTxt("twister:TooManyOutputs", "Too many output arguments.");
201     }
202
203     /* No args given, return a scalar.
204     */
205     if (nrhs == 0) {
206         plhs[0] = mxCreateDoubleScalar(genrand_res53());
207     }
208     else if ((nrhs > 0) && (mxIsChar(prhs[0]))) {
209         char theString[10], *p = theString;
210         if (mxGetString(prhs[0], theString, sizeof(theString))) {
211             mexErrMsgIdAndTxt("twister:BadStringArg", "Invalid string arg.");
212         }
213         for (; *p; p++) *p = tolower(*p); /* need strncasecmp */
214         if (strcmp(theString, "state", sizeof(theString))) {
215             mexErrMsgIdAndTxt("twister:BadStringArg", "Invalid string arg.");
216         }
217         else if (nrhs > 2) {
218             mexErrMsgIdAndTxt("twister:TooManyInputs", "Too many input arguments.");
219         }
220
221         /* Return the old state when setting the state only if asked for, but
222         * always return current state when reading the state.
223         */
224         if ((nlhs > 0) || (nrhs == 1)) {
225             plhs[0] = mxCreateNumericMatrix(1, N+1, mxUINT32_CLASS, mxREAL);
226             unsigned long *q = mt, *p = (unsigned long *)mxGetData(plhs[0]);
227             for (int i=N; i; i--) *p++ = *q++;
228             *p++ = mti;
229         }
230
231         /* Init or set the state.
232         */
233         if (nrhs == 2) {
234             unsigned long initLen = (unsigned long)mxGetNumberOfElements(prhs[1]);
235             if (initLen == 1) {
236                 /* M&N's default initializer (see genrand_int32) is
237                 * 5489UL, make zero do the same thing as that.
238                 */
239                 unsigned long initVal = (unsigned long)mxGetScalar(prhs[1]);
240                 if (initVal == 0) initVal = 5489UL;
241                 init_genrand(initVal);
242             }
243             else if (mxIsDouble(prhs[1])) {
244                 if (initLen <= N) {
245                     double *q = (double *)mxGetData(prhs[1]);
246                     unsigned long init[N], *p = init;
247                     for (int i=initLen; i; i--) *p++ = (unsigned long) *q++;
248                     init_by_array(init, initLen);
249                 }
250                 else {
251                     mexErrMsgIdAndTxt("twister:InvalidInitLen", "Initializer J must have fewer than
252                     625 elements.");
253                 }
254             }
255             else if (mxIsUint32(prhs[1])) {
256                 unsigned long stateLen = (unsigned long)mxGetNumberOfElements(prhs[1]);
257                 unsigned long *state = (unsigned long *)mxGetData(prhs[1]);
258                 if ((stateLen == N+1) && (state[N] <= N)) {
259                     unsigned long *q = state, *p = mt;
260                     for (int i=N; i; i--) *p++ = *q++;
261                     mti = (int) *q++;
262                 }
263                 else {
264                     mexErrMsgIdAndTxt("twister:InvalidStateLen", "Invalid state vector S.");
265                 }
266             }
267             else {
268                 mexErrMsgIdAndTxt("twister:InvalidInitOrState", "Second input must be an initializer
269                 or a state vector.");
270             }
271         }
272     }
273 }

```



```

264 } else {
265     int nelelem, localDims[10];
266     int *dims = localDims;
267     int ndim = (nrhs == 1) ? mxGetNumberOfElements(prhs[0]) : nrhs;
268
269     if (ndim > 10) {
270         dims = (int *)mxCalloc(ndim, sizeof(int));
271     }
272
273     /* Individual size args given.
274     */
275     if (nrhs > 1) {
276         int *p = dims;
277         nelelem = 1;
278         for (int i=0; i<ndim; i++) {
279             if ((!mxIsDouble(prhs[i])) || (mxGetNumberOfElements(prhs[i])!=1) || mxIsComplex(prhs[
280                 i])) {
281                 errMsgNum = 101; goto cleanup;
282             }
283             nelelem *= *p++ = (int)mxGetScalar(prhs[i]);
284         }
285
286         /* Size vector given.
287         */
288     } else { /* nrhs == 1, nrhs==0 has already been weeded out */
289         if ((!mxIsDouble(prhs[0])) || (mxGetNumberOfElements(prhs[0])<1) || mxIsComplex(prhs[0]))
290             {
291                 errMsgNum = 102; goto cleanup;
292             }
293
294         /* Single size given, we'll return a square matrix.
295         */
296         else if (ndim == 1) {
297             ndim = 2;
298             int n = (int)mxGetScalar(prhs[0]);
299             dims[0] = n; dims[1] = n;
300             nelelem = n*n;
301
302             /* Size vector.
303             */
304             else {
305                 int *p = dims;
306                 double *q = (double*)mxGetData(prhs[0]);
307                 nelelem = 1;
308                 for (int i=ndim; i; i--) nelelem *= *p++ = (int)*q++;
309             }
310         }
311
312         for (int i=0; i<ndim; i++) {
313             if ((dims[i] < 0) || dims[i] > INT_MAX) {
314                 errMsgNum = 103; goto cleanup;
315             }
316         }
317
318         /* Create the output matrix, get a pointer to its data, and fill it
319         * in with random values.
320         */
321         {
322             plhs[0] = mxCreateNumericArray(ndim, dims, mxDOUBLE_CLASS, mxREAL);
323             double *r = (double*) mxGetData(plhs[0]);
324             for (int i=nelelem; i; i--,r++) *r = genrand_res53();
325         }
326
327 cleanup:
328         if (dims != localDims) mxFree((void *)dims);
329         if (errMsgNum) {
330             mexErrMsgIdAndTxt("twister:InvalidSize", "Invalid output size.");
331         }
332     }
333 }

```